A Semantics for Propositions as Sessions

A Semantics for Propositions as Sessions

Sam Lindley and J. Garrett Morris

The University of Edinburgh
{Sam.Lindley,Garrett.Morris}@ed.ac.uk

Abstract. Session types provide a static guarantee that concurrent programs respect communication protocols. Recently, Caires, Pfenning, and Toninho, and Wadler, have developed a correspondence between propositions of linear logic and session typed π-calculus processes. We relate the cut-elimination semantics of this approach to an operational semantics for session-typed concurrency in a functional language. We begin by presenting a variant of Wadler's session-typed core functional language, GV. We give a small-step operational semantics for GV. We develop a suitable notion of deadlock, based on existing approaches for capturing deadlock in π-calculus, and show that all well-typed GV programs are deadlock-free, deterministic, and terminating. We relate GV to linear logic by giving translations between GV and CP, a process calculus with a type system and semantics based on classical linear logic. We prove that both directions of our translation preserve reduction; previous translations from GV to CP, in contrast, failed to preserve β-reduction. Furthermore, to demonstrate the modularity of our approach, we define two extensions of GV which preserve deadlock-freedom, determinism, and termination.

1 Introduction

From massively distributed programs running across entire data centres, to handheld apps reliant on remote services for functionality, concurrency has become a critical aspect of modern programs, and thus a central problem in program correctness. Assuring correct concurrent behaviour requires reasoning not just about the types of data communicated, but the order in which the communication takes place. For example, the messages between an SMTP client and server are all strings, but a client that sends the recipient's address before the sender's address is in violation of the protocol despite sending the correct type of data.

Session types, originally proposed by Honda [13], provide a mechanism to reason about the state of channel-based communication. The type of a channel captures the expected behaviour of a process communicating on that channel. For example, we might express a simplified session type for an SMTP client as:

$$!\mathit{FromAddress}.!\mathit{ToAddress}.!\mathit{Message}.\mathsf{end}$$

where $!T.S$ is the type of a channel that sends a value of type T, then continues with behaviour specified by S. An important feature of session types is duality: the session type of an SMTP server is the dual of the session type of the client:

$$?\mathit{FromAddress}.?\mathit{ToAddress}.?\mathit{Message}.\mathsf{end}$$

where $?T.S$ is the type of a channel that sends a value of type T, then continues with behaviour specified by S. Honda originally defined session types for process calculi; recent work [10, 25] has investigated the use of session types for concurrency in functional languages.

Session type systems are necessarily substructural—if processes can freely discard or duplicate channels, then the type system cannot guarantee that observable messages on channels match their expected types. Recent work seeks to establish a correspondence between session types and linear logic, an archetypal substructural logic for reasoning about state. Caires and Pfenning [5] develop a correspondence between cut elimination in intuitionistic linear logic and process reduction in a session-typed process calculus. Wadler [26] adapts their approach to classical linear logic, emphasising the role of duality in typing; the semantics of his system is given directly by the cut elimination rules of classical linear logic. He gives a type-preserving translation from a simple functional calculus (GV), inspired by Gay and Vasconcelos [10], to a process calculus (CP), inspired by Caires and Pfenning [5]. However, he gives no semantics for GV other than by translation to CP.

In this paper, we develop a session-typed functional core calculus, also called GV. (Our language shares most of the distinctive features of Wadler's, although it differs in some details.) We present a small-step operational semantics for GV, factored into functional and concurrent portions following the approach of Gay and Vasconcelos [10]. The functional portion of our semantics differs from standard presentations of call-by-value reduction only in that we adopt a weak form of explicit substitution to enable a direct correspondence with cut reduction. The concurrent portion of our semantics includes the typical reductions and equivalences of π-calculus-like process calculi.

Ultimately, our goal is to build and reason about functional programming languages extended with session types. Thus GV is a natural fit. Indeed, we are currently implementing an asynchronous variant of GV as part of the Links web programming language [8]. Developing a direct semantics for GV provides a number of benefits over relying on the translation to CP.

- It provides a simple semantic characterisation of deadlock. Unlike Wadler's proof of deadlock freedom, ours does not depend on normalisation, and thus extends to non-terminating processes.
- The proof technique itself is modular: as illustrated by the extensions, the same technique can be applied to practical (sometimes non-logical) extensions of the language.
- Compared to cut-elimination in CP, the GV semantics is much closer to something one might actually want to implement in practice, as witnessed by our Links implementation.

We believe in modularity, and so re-use as much of the standard linear lambda calculus machinery as possible, while limiting non-standard extensions. The paper proceeds as follows.

- We define a core linearly-typed functional language, GV, by extending linear lambda calculus with session-typed communication primitives (§2.1).

Session types	$S ::= !T.S \mid ?T.S \mid \mathsf{end}_! \mid \mathsf{end}_? \mid S^\sharp$
Types	$T, U ::= S \mid \mathbf{1} \mid T \times U \mid \mathbf{0} \mid T + U \mid T \multimap U$
Terms	$L, M, N ::= x \mid K\ M \mid \lambda x.M \mid M\ N$
	$\mid\ (M, N) \mid \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N$
	$\mid\ \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{case}\ M\ \{\mathsf{inl}\ x \mapsto N; \mathsf{inr}\ x \mapsto N\}$
	$\mid\ () \mid \mathsf{let}\ () = M\ \mathsf{in}\ N \mid \mathsf{absurd}\ M$
Constants	$K ::= \mathsf{send} \mid \mathsf{receive} \mid \mathsf{fork} \mid \mathsf{wait} \mid \mathsf{link}$

Fig. 1: Syntax of GV Terms and Types

We present an (untyped) synchronous operational semantics for GV (§2.2). We characterise deadlock and normal forms; we show that typed terms are deadlock-free, that closed typed terms evaluate to normal forms (§2.3), and that evaluation is deterministic and terminating (§2.4).

– We connect GV to the interpretation of session types as linear logic propositions, by establishing a correspondence between the semantics of GV and that of CP. We begin by introducing CP (§3.1). We show that we can simulate CP reduction in GV (§3.2), and GV reduction in CP (§3.3). (As π-calculus-like process calculi provide substitution only for names, not entire process expressions, the latter depends crucially on the use of weak explicit substitutions in the semantics of GV lambda abstractions.)

– We consider two extensions of GV: one which has a single self-dual type for closed channels, harmonising the treatment of closed channels with that of other session-typed calculi (§4.1), and another which adds unlimited types and replicated behaviour (§4.2). We show that these extensions preserve the essential meta theoretic properties of the core language.

We conclude by discussing related (§5) and future (§6) work.

2 A Session-Typed Functional Language

2.1 Syntax and Typing

Figure 1 gives the syntax of GV types and terms. The types T include nullary ($\mathbf{0}$) and binary ($T + U$) linear sums, nullary ($\mathbf{1}$) and binary ($T \times U$) linear products, and linear implication ($T \multimap U$). We frequently write $M; N$ as the elimination form of $\mathbf{1}$ in place of the more verbose $\mathsf{let}\ () = M\ \mathsf{in}\ N$. Session types S include input ($?T.S$), output ($!T.S$), and closed channels ($\mathsf{end}_!$, $\mathsf{end}_?$). We also include a type $S^\sharp$ of channels; values of channel type cannot be used directly in terms, but will appear in the typing of thread configurations. The terms are the standard λ-calculus terms, augmented with constructs for pairs and sums. Figure 2 gives both typing rules and type schemas for the constants. Note that core GV judgements are linear, i.e., not subject to weakening or contraction.

Concurrency. Concurrent behaviour is provided by the constants. Communication is provided by send and receive. For example (assuming an extension of

Typing rules

$$\frac{T \neq S^\sharp}{x : T \vdash x : T} \qquad \frac{K : T \multimap U \quad \Gamma \vdash M : T}{\Gamma \vdash K\ M : U}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \multimap U} \qquad \frac{\Gamma \vdash M : T \multimap U \quad \Gamma' \vdash N : T}{\Gamma, \Gamma' \vdash M\ N : U}$$

$$\frac{\Gamma \vdash M : T \quad \Gamma' \vdash N : U}{\Gamma, \Gamma' \vdash (M, N) : T \times U} \qquad \frac{\Gamma \vdash M : T \times T' \quad \Gamma', x : T, y : T' \vdash N : U}{\Gamma, \Gamma' \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N : U}$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \mathsf{inl}\, M : T + U} \qquad \frac{\Gamma \vdash M : T + T' \quad \Gamma', x : T \vdash N : U \quad \Gamma', x : T' \vdash N' : U}{\Gamma, \Gamma' \vdash \mathsf{case}\ M\ \{\mathsf{inl}\, x \mapsto N; \mathsf{inr}\, x \mapsto N'\} : U}$$

$$\frac{}{\vdash () : \mathbf{1}} \qquad \frac{\Gamma \vdash M : \mathbf{1} \quad \Gamma' \vdash N : T}{\Gamma, \Gamma' \vdash \mathsf{let}\ () = M\ \mathsf{in}\ N : T} \qquad \frac{\Gamma \vdash M : \mathbf{0}}{\Gamma, \Gamma' \vdash \mathsf{absurd}\, M : T}$$

Type schemas for constants

$$\mathsf{send} : T \times !T.S \multimap S \qquad \mathsf{receive} : ?T.S \multimap T \times S \qquad \mathsf{fork} : (S \multimap \mathsf{end}_!) \multimap \overline{S}$$

$$\mathsf{wait} : \mathsf{end}_? \multimap \mathbf{1} \qquad \mathsf{link} : S \times \overline{S} \multimap \mathsf{end}_!$$

Duality

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S} = !T.\overline{S} \qquad \overline{\mathsf{end}_!} = \mathsf{end}_? \qquad \overline{\mathsf{end}_?} = \mathsf{end}_!$$

Fig. 2: GV Typing Rules

our core language with numbers and arithmetic operators), a program M that receives a pair of numbers along channel z and then sends their sum along the same channel can be expressed as

$$M \triangleq \mathsf{let}\ ((x, y), z) = \mathsf{receive}\ z\ \mathsf{in}\ \mathsf{send}\ (x + y, z)$$

(where the interpretation of nested patterns by sequences of bindings is standard). Channels are treated linearly in GV. Thus, receive returns not only the received value (the pair of x and y) but also a new copy of the channel used for receiving z; similarly, send returns a copy of the channel used for sending. Thus, the term above is well-typed in the context $z : ?(Int \times Int).!Int.S$, and evaluates to a channel of type S. Session initiation is provided by fork. If f is a function from a channel of type S to a closed channel (of type $\mathsf{end}_!$), then $\mathsf{fork}\, f$ forks a new thread in which f is applied to a fresh channel of type S, and returns a channel of type $\overline{S}$ in order to communicate with the thread. For example, the term $\mathsf{fork}\ (\lambda z.M)$ returns a channel of type $!(Int \times Int).?Int.\mathsf{end}_?$. Given a thread created by $\mathsf{fork}\, f$, the channel returned from f is closed by fork, whereas the other end of the channel must be closed by calling wait. A client of

the process M can be defined as follows:

$$N \triangleq \mathsf{let}\ z = \mathsf{send}\,((6,7),z)\ \mathsf{in}\ \mathsf{let}\ (x,z) = \mathsf{receive}\,z\ \mathsf{in}\ \mathsf{wait}\,z; x$$

The combined process $\mathsf{let}\ x = \mathsf{fork}\,(\lambda z.M)\ \mathsf{in}\ N$ evaluates to 13. The expression $\mathsf{link}\,(x,y)$ forwards messages sent on x to be received on y and vice versa. We choose to include it as a primitive as it corresponds to the axiom rule of linear logic, which is standard in logical accounts of session types.

Choice. In addition to input and output, typical session type systems also provide session types representing internal ($S_1 \oplus S_2$) and external ($S_1 \,\&\, S_2$) choice (also known as selection and branching, respectively). For example, we might write a process that can either sum two numbers or negate one:

$$\begin{array}{l}\mathsf{offer}\ z\ \{\ \mathsf{inl}\ z \mapsto \mathsf{let}\ ((x,y),z) = \mathsf{receive}\,z\ \mathsf{in}\ \mathsf{send}\,(x+y,z)\\ \qquad\ \ \mathsf{inr}\ z \mapsto \mathsf{let}\ (x,z) = \mathsf{receive}\,z\ \mathsf{in}\ \mathsf{send}\,(-x,z)\ \}\end{array}$$

This term initially requires $z : (?(Int \times Int).!Int.S) \,\&\, (?Int.!Int.S)$. A client of this process begins by choosing which branch of the session to take; for example, we can extend the preceding example as follows:

$$\mathsf{let}\ z = \mathsf{select}\ \mathsf{inl}\ z\ \mathsf{in}\ \mathsf{let}\ z = \mathsf{send}\,((6,7),z)\ \mathsf{in}\ \mathsf{let}\ (x,z) = \mathsf{receive}\,z\ \mathsf{in}\ \mathsf{wait}\,z; x$$

While we would expect a surface language to include selection and branching, we omit them from our core calculus. Instead, we show that they are macro-expressible using the linear sum type. The intuition is that selection is implemented by sending a suitably tagged process, while branching is implemented by a term-level branch on a received value. Concretely, we define the types by:

$$S_1 \oplus S_2 \triangleq\ !(\overline{S_1} + \overline{S_2}).\mathsf{end}_! \qquad\qquad S_1 \,\&\, S_2 \triangleq\ ?(S_1 + S_2).\mathsf{end}_?$$

Note that we have the expected duality relationship: $\overline{S_1} \oplus \overline{S_2} = \overline{S_1 \,\&\, S_2}$. We implement select and offer as follows (where ℓ ranges over $\{\mathsf{inl},\mathsf{inr}\}$):

$$\begin{array}{rl}\mathsf{select}\,\ell\,M \triangleq & \mathsf{fork}(\lambda x.\mathsf{send}\,(\ell\,x, M))\\ \mathsf{offer}\,M\,\{\mathsf{inl}\,x \mapsto P; \mathsf{inr}\,x \mapsto Q\} \triangleq & \mathsf{let}\ (x,y) = \mathsf{receive}\,M\ \mathsf{in}\\ & \mathsf{wait}\,y; \mathsf{case}\,x\,\{\mathsf{inl}\,x \mapsto P; \mathsf{inr}\,x \mapsto Q\}\end{array}$$

Correspondingly, nullary choice and selection are encoded using the $\mathbf{0}$ type:

$$\oplus\{\} \triangleq\ !\mathbf{0}.\mathsf{end}_! \qquad\qquad \&\,\{\} \triangleq\ ?\mathbf{0}.\mathsf{end}_?$$

$$\mathsf{offer}\,M\,\{\} \triangleq \mathsf{let}\ (x,y) = \mathsf{receive}\,M\ \mathsf{in}\ \mathsf{wait}\,y; \mathsf{absurd}\{\}$$

2.2 Semantics

Following Gay and Vasconcelos [10], we factor the semantics of GV into a (deterministic) reduction relation on terms (called $\longrightarrow_{\mathrm{V}}$) and a (non-deterministic) reduction on configurations of processes (called $\longrightarrow$). Figure 3 gives the syntax of values, configurations, and evaluation and configuration contexts.

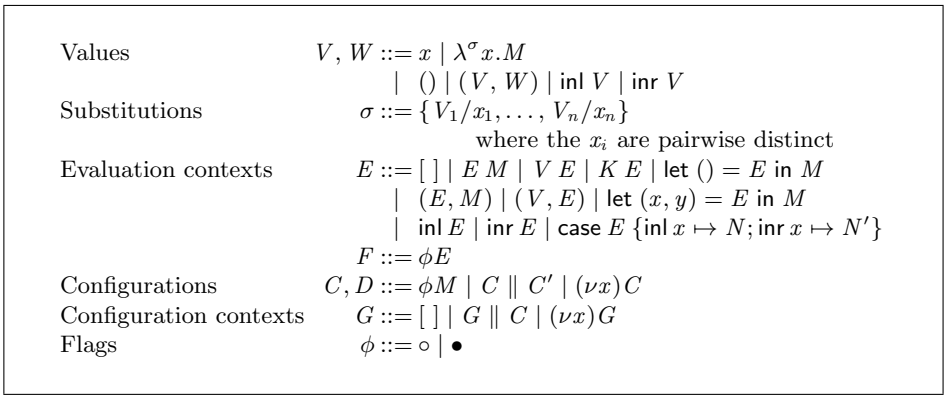

Fig. 3: Syntax of Values, Configurations, and Contexts

Terms. To preserve a close connection between the semantics of our term language and cut-reduction in linear logic, we define term reduction using weak explicit substitutions [18]. In this approach, we intercept substitutions at λ-terms rather than immediately applying them to the body of the term. Thus, our language of terms includes closures $\lambda^\sigma x.M$, where σ provides the intercepted substitution. We extend the typing judgement to include closures, as follows:

$$\frac{\Gamma, x : T \vdash M\sigma : U \quad dom(\sigma) = (fv(M) \setminus \{x\})}{\Gamma \vdash \lambda^\sigma x.M : T \multimap U}$$

The free variables of a closure $\lambda^\sigma x.M$ are the free variables of the range of σ, not the free variables of M. The capture avoiding substitution $M\sigma$ of σ applied to M is defined as usual on the free variables of M. Note that the side condition on the domain of σ is preserved under substitution. We implicitly treat plain lambda abstractions $\lambda x.M$ as closures $\lambda^\sigma x.M$, where σ is a renaming substitution restricted to the free variables of M less $\{x\}$; concretely:

$$\begin{array}{l} \lambda x.M \triangleq \lambda^\sigma x.(M\sigma') \\ \quad \text{where } fv(M) \setminus \{x\} = \{x_1, \ldots, x_n\} \qquad y_1, \ldots, y_n \text{ are fresh variables} \\ \qquad\quad \sigma = \{x_1/y_1, \ldots, x_n/y_n\} \qquad\quad \sigma' = \{y_1/x_1, \ldots, y_n/x_n\} \end{array}$$

We lift the typing judgement on terms pointwise to substitutions:

$$\frac{\Gamma_1 \vdash \sigma(x_1) : \Delta(x_1) \cdots \Gamma_k \vdash \sigma(x_k) : \Delta(x_k) \quad dom(\sigma) = dom(\Delta)}{\Gamma_1, \ldots, \Gamma_k \vdash \sigma : \Delta}$$

Configurations. The grammar of configurations includes the usual π-calculus forms for composition and name restriction. However, because functional computations return values (which may, in turn, contain channels), we distinguish between the "main" thread $\bullet M$ (which returns a value) and the threads $\circ M$ created by fork (which do not).

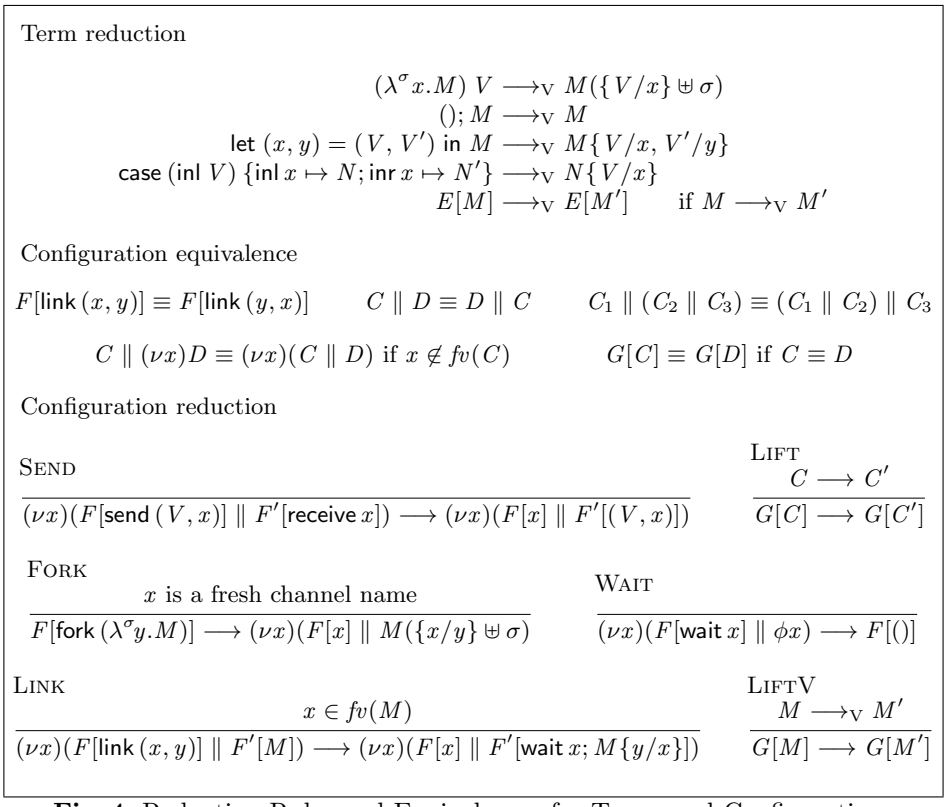

Fig. 4: Reduction Rules and Equivalences for Terms and Configurations

Reduction. Reduction rules for terms and configurations, and equivalences for configurations, are given in Figure 4. Term reduction ($\longrightarrow_{\mathrm{V}}$) implements call-by-value, left-to-right evaluation. Configuration equivalence ($\equiv$) is standard. Communication is provided by Send and session initiation by Fork. Rule Wait combines synchronisation of closed channels with garbage collection of the associated name restriction. Rule Link is complicated by the need to produce a channel of type $\mathsf{end}_!$; the inserted wait synchronises with the produced channel.

Relation Notation. We write $R\,R'$ for sequential composition and $R \cup R'$ for union of R and R'. We write R^+ for transitive closure and $R^\star$ for the reflexive, transitive closure of R.

Configuration Typing. Our syntax of configurations permits various forms of deadlocked configurations. For example, if we define the terms M and N by

$$M \triangleq \mathsf{let}\ (z, y) = \mathsf{receive}\ y\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{send}\,(z, x)\ \mathsf{in}\ M' \qquad N \triangleq \mathsf{let}\ (z, x) = \mathsf{receive}\ x\ \mathsf{in}\ \mathsf{let}\ y = \mathsf{send}\,(z, y)\ \mathsf{in}\ N'$$

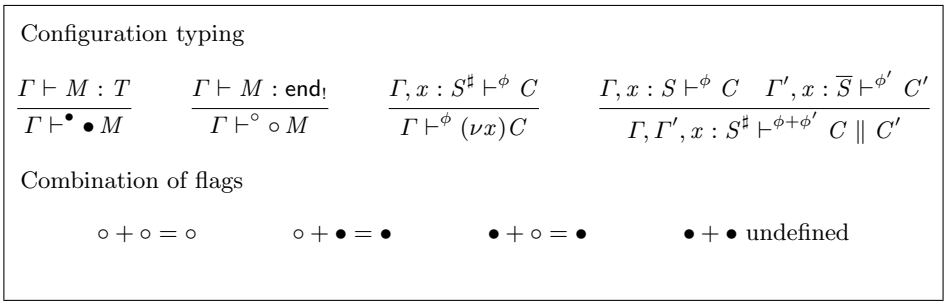

Fig. 5: Configuration Typing

given suitable terms M' and N', then it is apparent that configurations such as $(\nu xy)M$, $(\nu xy)(M \parallel M)$ and $(\nu xy)(M \parallel N)$ cannot reduce further, even though M and N can be individually well-typed. To exclude such cases, we provide a type discipline for configurations (Figure 5). It is based on type systems for linear π-calculus [17], but with two important differences.

- First, we seek to assure that there is at most one main thread. This constraint is enforced by the flags ($\bullet$ and $\circ$) on the derivations: a derivation $\Gamma \vdash^{\bullet} C$ indicates that configuration C contains the main thread, while $\Gamma \vdash^{\circ} C$ indicates that C does not contain the main thread. We write $\Gamma \vdash C$ to abbreviate $\exists \phi. \Gamma \vdash^{\phi} C$, that is, C may include a main thread.
- Second, we require that exactly one channel is shared at each composition of processes. This is more restrictive than standard type systems for linear π-calculus, which allow an arbitrary number of channels (including none) to be shared at a composition of processes.

Notice that the above stuck examples are ill-typed in this system: $(\nu xy)M$ because y must have a type $S^\sharp$ in M; $(\nu xy)(M \parallel M)$ because there is no type $S^\sharp$ such that both S and $\overline{S}$ are of the form $?T.S'$, as required by receive; and, $(\nu xy)(M \parallel N)$ because both x and y must be shared between M and N, but the typing rule for composition only allows one channel to be shared.

Now we can show that reduction preserves typing. We begin with terms.

Lemma 1. *If $\Gamma \vdash M : T$ and $M \longrightarrow_V M'$, then $\Gamma \vdash M' : T$*

The proof is by induction on M; the cases are all standard. We now extend this result to configurations.

Theorem 2. *If $\Gamma \vdash C$ and $C \longrightarrow C'$ then $\Gamma \vdash C'$.*

The proof is by induction on the derivation of $C \longrightarrow C'$; the cases are given in Appendix A.

Typing and Configuration Equivalence. Alas, our notion of typing is not preserved by configuration equivalence. For example, assume that $\Gamma \vdash (\nu xy)(C_1 \parallel (C_2 \parallel C_3))$, where $x \in fv(C_1), y \in fv(C_2)$, and $x, y \in fv(C_3)$. We have that

$C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3$, but $\Gamma \nvdash (\nu xy)((C_1 \parallel C_2) \parallel C_3)$, as both x and y must be shared between the processes $C_1 \parallel C_2$ and C_3. However, we can show that starting from a well-typed configuration, we need never rely on an ill-typed equivalent configuration to expose possible reductions.

Theorem 3. *If $\Gamma \vdash C$, $C \equiv C'$ and $C' \longrightarrow D'$, then there exists D such that $D \equiv D'$, and $\Gamma \vdash D$.*

Proof. Observe that if $\Gamma \vdash C$, then for any pair of terms M_1, M_2 appearing in C, there are environments Γ_1, Γ_2 and types T_1, T_2 such that $\Gamma_1 \vdash M_1 : T_1, \Gamma_2 \vdash M_2 : T_2$, and (because of the typing rule for composition) Γ_1 and Γ_2 share at most one variable. By examination of the reduction rules, we can conclude that there are well-typed C_0, D_0 such that $C' = G[C_0]$, $C_0 \longrightarrow D_0$ and $D' = G[D_0]$. The result then follows by structural induction on C, examining the possible equivalences in each case. □

We extend Theorem 3 to sequences of reductions, defining $\Longrightarrow$ as $(\equiv \longrightarrow \equiv)^\star$.

Corollary 4. *If $\Gamma \vdash C$ and $C \Longrightarrow D$, then there exists D' such that $D \equiv D'$, and $\Gamma \vdash D'$.*

2.3 Deadlock and its Absence

In the previous section, we saw examples of deadlocked terms which were rejected by our type system. We now present a general account of deadlock: we characterise deadlocked configurations, and show that well-typed configurations do not evaluate to deadlocked configurations.

We begin by observing that many examples of stuck configurations are already excluded by existing session-typing disciplines: in particular, those configurations in which either too many or too few threads attempt to synchronise on a given channel, or those with inconsistent use of channels. The cases of interest to us are those in which the threads individually obey the session-typing discipline, but the order of synchronisation in the threads creates deadlock. We say that a thread M is blocked on a channel x, written $\mathsf{blocked}(x, M)$, if M has evaluated to some context surrounding a communication primitive applied to x:

$$\mathsf{blocked}(x, M) \stackrel{\text{def}}{\iff} \exists N.\, M = E[\mathsf{send}\,(N, x)] \vee M = E[\mathsf{receive}\,x] \vee M = E[\mathsf{wait}\,x]$$

In such a case, M can only reduce further in composition with another thread blocked on x, and any communication on other channels in M will be delayed until a communication on x has occurred. We abstract over the property that y depends on x in M, abbreviated $\mathsf{depends}(x, y, M)$; in other words, M is blocked on x, but has y as one of its (other) free variables. We extend this notion of dependency from single threads to configurations of threads, with the observation that in a larger configuration intermediate channels may participate in the dependency.

$$\begin{aligned} \mathsf{depends}(x, y, E[M]) &\stackrel{\text{def}}{\iff} \mathsf{blocked}(x, M) \wedge y \in fv(E[M]) \\ \mathsf{depends}(x, y, C) &\stackrel{\text{def}}{\iff} (C \equiv G[M] \wedge \mathsf{depends}(x, y, M)) \vee (C \equiv G[D \parallel D'] \\ &\qquad \wedge (\exists z.\mathsf{depends}(x, z, D) \wedge \mathsf{depends}(z, y, D'))) \end{aligned}$$

We now define deadlocked configurations as those with cyclic dependencies:

$$\mathsf{deadlocked}(C) \overset{\text{def}}{\iff} C \equiv G[D \parallel D'] \wedge \exists x, y.\mathsf{depends}(x, y, D) \wedge \mathsf{depends}(y, x, D')$$

Because the definition of dependency permits intermediate channels, this definition encompasses cycles involving an arbitrary number of channels. We say that a configuration C is deadlock free if, for all D such that $C \Longrightarrow D$, $\neg\mathsf{deadlocked}(D)$. Observe that if $C \equiv D$, $\mathsf{deadlocked}(C) \iff \mathsf{deadlocked}(D)$.

At this point, we can observe that in any deadlocked configuration there must be a composition of configurations that shares more than one channel. This is precisely the situation that is excluded by our configuration type system.

Lemma 5. *If $\Gamma \vdash C$, and $C = G[D \parallel D']$, then $fv(D) \cap fv(D') = \{x\}$ for some variable x.*

Proof. By structural induction on the derivation of $\Gamma \vdash C$; the only interesting case is for parallel composition, where the desired result is assured by the partitioning of the environment. □

To extend this observation to deadlock freedom, we must take equivalence into account. While it is true that equivalence need not preserve typing, there are no equivalence rules that affect the free variables of individual threads. Thus, cycles of dependent channels are preserved by equivalence.

Lemma 6. *If $\Gamma \vdash C$ then $\neg\mathsf{deadlocked}(C)$.*

Proof. By contradiction. Suppose $\mathsf{deadlocked}(C)$, then by expanding the definition of $\mathsf{deadlocked}$ we know that there must exist variables $x_1, \ldots, x_n$ and processes $M_1, \ldots, M_n$ in C such that:

$$\mathsf{depends}(x_1, x_2, M_1) \wedge \mathsf{depends}(x_2, x_3, M_2) \wedge \cdots \wedge \mathsf{depends}(x_n, x_1, M_n)$$

Either $n = 1$, which violates linearity, or configuration C must partition the cycle. However, any cut of the cycle is crossed by at least two channels, so C must be ill-typed by Lemma 5. □

Finally, we can combine the previous result with preservation of typing to show that well-typed terms never evaluate to deadlocked configurations.

Theorem 7. *If $\Gamma \vdash M : T$, then $\bullet\, M$ is deadlock-free.*

Proof. If $\Gamma \vdash M : T$, then $\Gamma \vdash \bullet\, M$, and so $\neg\mathsf{deadlocked}(\bullet\, M)$ and, for any D such that $\bullet\, M \Longrightarrow D$, we know that there is a well-typed $D' \equiv D$, and so $\neg\mathsf{deadlocked}(D)$. □

Progress and Canonical Forms. We conclude this section by describing a canonical form for configurations, and characterising the stuck terms resulting from the evaluation of well-typed terms. One might hope that evaluation of a well-typed term would always produce a value; however, this is complicated because terms may return channels. For a simple example, consider the term:

$$\bullet\,\mathsf{fork}\,(\lambda x.\mathsf{let}\ (y,x) = \mathsf{receive}\,x\ \mathsf{in}\ \mathsf{send}\,(y,x))$$

This term spawns a thread (which simply echoes once), and then returns the resulting channel; thus, the result of evaluation is a configuration equivalent to:

$$(\nu x)(\bullet\,x \parallel \circ\,\mathsf{let}\ (y,x) = \mathsf{receive}\,x\ \mathsf{in}\ \mathsf{send}\,(y,x))$$

Clearly, no more evaluation is possible, even though the configuration still contains blocked threads. However, it turns out that we can show that evaluation of terms that do not return channels must always produce a value (Corollary 12).

Definition 8. *A process C is in canonical form if there is some sequence of variables $x_1, \ldots, x_{n-1}$ and terms $M_1, \ldots, M_n$ such that:*

$$C = (\nu x_1)(\circ\, M_1 \parallel (\nu x_2)(\circ\, M_2 \parallel \cdots \parallel (\nu x_{n-1})(\circ\, M_{n-1} \parallel \phi M_n)\ldots))$$

Note that canonical forms need not be unique. For example, consider the configuration $\vdash (\nu xy)(C_1 \parallel C_2 \parallel C_3)$ where $x \in fv(C), y \in fv(D)$, and $x, y \in fv(C_3)$. Both $(\nu x)(C_1 \parallel (\nu y)(C_2 \parallel C_3))$ and $(\nu y)(C_2 \parallel (\nu x)(C_1 \parallel C_3))$ are canonical forms of the original configuration. We can show that any well-typed term must be equivalent to a term in canonical form; again, the key insight is that captured by Lemma 5: if any two sub-configurations share at most one channel, then we can order the threads by the channels they share.

Lemma 9. *If $\Gamma \vdash C$, then there is some $C' \equiv C$ such that $\Gamma \vdash C'$ and C' is in canonical form.*

The proof is by induction on the count of bound variables; the details are in Appendix A.

We can now state some progress results. We begin with open configurations: each thread must be blocked on either a free variable or a ν-bound variable.

Theorem 10. *Let $\Gamma \vdash C$, $C \not\longrightarrow$ and let $C' = (\nu x_1)(\circ\, M_1 \parallel (\nu x_2)(\circ\, M_2 \parallel \cdots \parallel (\nu x_{n-1})(\circ\, M_{n-1} \parallel \phi M_n)\ldots))$ be a canonical form of C. Then:*

1. *For $1 \leq i \leq n-1$ either $\mathsf{blocked}(x_j, M_i)$ where $j \leq i$ or $\mathsf{blocked}(y, M_i)$ for some $y \in dom(\Gamma)$; and,*
2. *Either M_n is a value or $\mathsf{blocked}(y, M_n)$ for some $y \in \{x_i \mid 1 \leq i \leq n-1\} \cup dom(\Gamma)$.*

Proof. By induction on the derivation of $\Gamma \vdash C'$, using the definition of $\longrightarrow$. □

We can strengthen the result significantly when we move to configurations without free variables. To see why, consider just the first two threads of a configuration $(\nu x_1)(M_1 \parallel (\nu x_2)(M_2 \parallel \dots))$. As there are no free variables, thread M_1 can only be blocked on x_1. Now, from the previous result, thread M_2 can be blocked on either x_1 or x_2. But, were it blocked on x_1, it could reduce with thread M_1; we can conclude it is blocked on x_2. Generalising this observation gives the following progress result.

Theorem 11. *Let $\vdash C$, $C \not\longrightarrow$ and let $C' = (\nu x_1)(\circ\, M_1 \parallel (\nu x_2)(\circ\, M_2 \parallel \cdots \parallel (\nu x_{n-1})(\circ\, M_{n-1} \parallel \phi M_n)\dots))$ be a canonical form of C. Then:*

1. *For $1 \leq i \leq n-1$, $\mathsf{blocked}(x_i, M_i)$; and,*
2. *M_n is a value.*

Proof. By induction on the derivation of $\vdash C'$, relying on Theorem 10. □

Finally, observe that some subset of the variables $x_1, \dots, x_n$ must appear in the result V. Therefore, if the original expression returns a value that does not contain any channels, it will evaluate to a configuration with no blocked threads (i.e., a single value).

Corollary 12. *Let $\vdash C$, $C \not\longrightarrow$ and C' be a canonical form of C such that the value returned by C' contains no channels, then $C' = \phi V$ for some value V.*

2.4 Determinism and Termination

It is straightforward to show that GV is deterministic. In fact, GV enjoys a strong form of determinism, called the *diamond property* [2].

Theorem 13. *If $\Gamma \vdash C$, $C \equiv\!\!\longrightarrow\!\!\equiv D_1$, and $C \equiv\!\!\longrightarrow\!\!\equiv D_2$, then either $D_1 \equiv D_2$ or there exists D_3 such that $D_1 \equiv\!\!\longrightarrow\!\!\equiv D_3$, and $D_2 \equiv\!\!\longrightarrow\!\!\equiv D_3$.*

Proof. First, observe that $\longrightarrow_{\mathrm{V}}$ is deterministic, and furthermore configuration reductions always treat $\longrightarrow_{\mathrm{V}}$ redexes linearly. This means we need only consider the interaction between different configuration reductions. Linear typing ensures that two configuration reductions cannot overlap. Furthermore, each configuration reduction is linear in the existing redexes, so we can straightforwardly perform the reductions in either order. □

It is not hard to see that the system remains deterministic if we extend the functional part of GV with any well-typed confluent reduction rules at all.

Theorem 14 (Strong normalisation). *If $\Gamma \vdash C$, then there are no infinite $\equiv\!\!\longrightarrow\!\!\equiv$ reduction sequences beginning from C.*

To prove strong normalisation for core GV, one can use an elementary argument based on linearity. When we add replication (§4.2) and other features, a logical relations argument along the lines of that of Perez et al. [21] suffices. Weak normalisation (the existence of a finite reduction sequence to an irreducible configuration) also follows as a direct corollary of Theorem 23 and the cut-elimination theorem for classical linear logic.

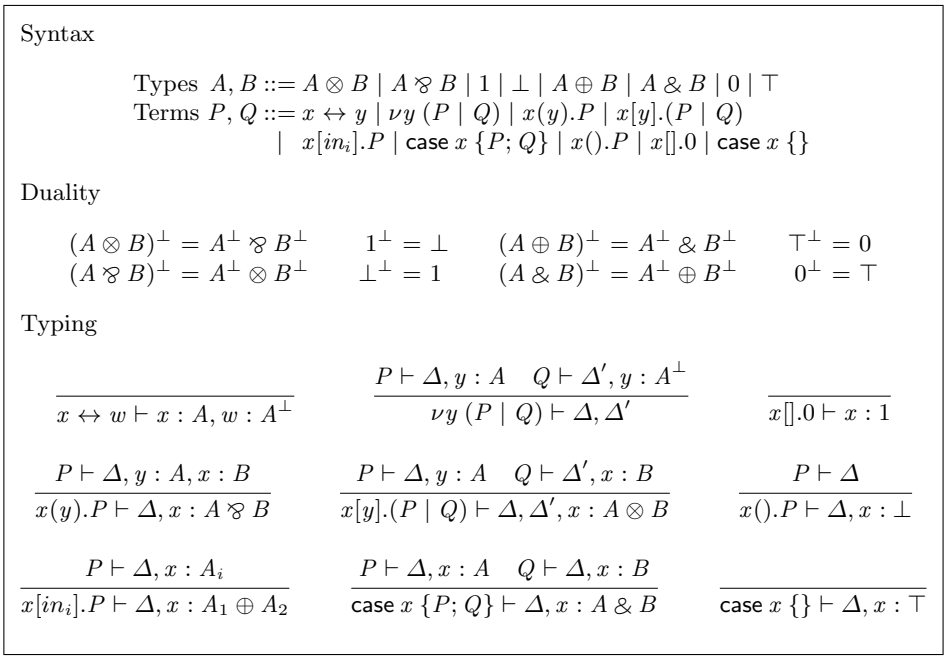

Fig. 6: CP Syntax and Typing

3 Classical Linear Logic

3.1 The Process Calculus CP

Figure 6 gives the syntax and typing rules for the multiplicative-additive fragment of CP; we let Δ range over typing environments. CP types and duality are the standard propositions and duality function of classical linear logic, while the terms are based on a subset of the π-calculus. The types $\&$ and $\oplus$ are interpreted as external and internal choice; the types $\parr$ and $\otimes$ are interpreted as input and output, while their units $\bot$ and 1 are interpreted as nullary input and output. Note that CP's typing rules implicitly rebind identifiers: for example, in the hypothesis of the rule for $\parr$, x identifies a proof of B, while in the conclusion it identifies a proof of $A \parr B$.

CP includes two rules that are logically derivable: the axiom rule, which is interpreted as channel forwarding, and the cut rule, which is interpreted as process composition. Two of CP's terms differ from standard π-calculus terms. The first is composition—rather than having distinct name restriction and composition operators, CP provides one combined operator. This syntactically captures the restriction that composed processes must share exactly one channel. The second is output: the CP term $x[y].(P \mid Q)$ includes output, composition, and name restriction (the name y designates a new channel, bound in P).

A Simpler Send The CP send rule is appealing because if one erases the terms it is exactly the classical linear logic rule for tensor. However, this correspondence comes at a price. Operationally, the process $x[y].(P \mid Q)$ does three things: it introduces a fresh variable y, it sends y to a freshly spawned process P, and in parallel it continues as process Q. This complicates both the reduction semantics of CP (as the cut reduction of $\otimes$ against $\parr$ must account for all three behaviours) and the equivalence of CP and GV (where the behaviour of send is simpler).

Following Boreale [4], we can give an alternative formulation of send, avoiding the additional name restriction and composition, as follows:

$$\frac{P \vdash \Delta, x : B, y : A}{x\langle y\rangle.P \vdash \Delta, x : A \otimes B, y : A^{\perp}}$$

where $x\langle y\rangle.P$ is defined as $x[z].(y \leftrightarrow z \mid P)$. In particular, note that

$$\begin{aligned} \nu x\,(x\langle y\rangle.P \mid x(z).Q) &= \nu x\,(x[z].(y \leftrightarrow z \mid P) \mid x(z).Q) \\ &\longrightarrow_{\mathrm{C}} \nu z\,(y \leftrightarrow z \mid \nu x\,(P \mid Q)) \\ &\longrightarrow_{\mathrm{C}} \nu x\,(P \mid Q\{y/z\}) \end{aligned}$$

as we would expect for synchronising a send and a receive. Similarly, we note that any process $x[y].(P \mid Q)$ can also be expressed as a process $\nu y\,(P \mid x\langle y\rangle.Q)$, which reduces to the original by one application of the commuting conversions. However, the two formulations are not quite identical. Let us consider the possible reductions of the two terms. Notice that in $x[y].(P \mid Q)$, both P and Q are blocked on x; however, the same is not true for $\nu y\,(P \mid x\langle y\rangle.Q)$; the latter permits reductions in P before synchronising on x.

Cut Elimination. The semantics of CP terms are given by cut reduction, as shown in Figure 7. We write $fv(P)$ for the free names of process P. Terms are identified up to structural congruence $\equiv$ (as name restriction and composition are combined into one form, composition is not always associative). We write $\longrightarrow_{\mathrm{C}}$ for the cut reduction relation, $\longrightarrow_{\mathrm{CC}}$ for the commuting conversion relation, and $\longrightarrow$ for $\longrightarrow_{\mathrm{C}} \cup \longrightarrow_{\mathrm{CC}}$. The majority of the cut reduction rules correspond closely to synchronous reductions in π-calculus—for example, the reduction of $\&$ against $\oplus$ corresponds to the synchronisation of an internal and external choice. The rule for reduction of $\parr$ against $\otimes$ is more complex than synchronisation of input and output in GV, as it must also manipulate the implicit name restriction and composition in CP's output term. We write $\Longrightarrow$ for $(\equiv\!\!\longrightarrow\!\!\equiv)^{+}$, $\Longrightarrow_{\mathrm{C}}$ for $(\equiv\!\!\longrightarrow_{\mathrm{C}}\!\!\equiv)^{+}$, and $\Longrightarrow_{\mathrm{CC}}$ for $\Longrightarrow_{\mathrm{C}} \longrightarrow^{\star}_{\mathrm{CC}}$.

Just as cut elimination in logic ensures that any proof may be transformed into an equivalent cut-free proof, the reduction rules of CP transform any term into a term blocked only on external communication—that is to say, if $P \vdash \Delta$, then $P \Longrightarrow_{\mathrm{CC}} P'$ where $P' \neq \nu x\,(Q \mid Q')$ for any x, Q, Q'. The final commuting conversions play a central role in this transformation, moving any remaining internal communication after an external communication. However, note that the commuting conversions do not correspond to computational steps (i.e., any reduction rule in π-calculus).

Structural congruence

$$
\begin{aligned}
x \leftrightarrow w &\equiv w \leftrightarrow x \\
\nu y\,(P \mid Q) &\equiv \nu y\,(Q \mid P) \\
\nu y\,(P \mid \nu z\,(Q \mid R)) &\equiv \nu z\,(\nu y\,(P \mid Q) \mid R), \quad \text{if } y \notin fv(R) \\
\nu x\,(P_1 \mid Q) &\equiv \nu x\,(P_2 \mid Q), \quad \text{if } P_1 \equiv P_2
\end{aligned}
$$

Primary cut reduction rules

$$
\begin{aligned}
\nu x\,(w \leftrightarrow x \mid P) &\longrightarrow_{\mathrm{C}} P[w/x] \\
\nu x\,(x[y].(P \mid Q) \mid x(y).R) &\longrightarrow_{\mathrm{C}} \nu x\,(Q \mid \nu y\,(P \mid R)) \\
\nu x\,(x[].0 \mid x().P) &\longrightarrow_{\mathrm{C}} P \\
\nu x\,(x[in_i].P \mid \mathsf{case}\ x\ \{Q_1; Q_2\}) &\longrightarrow_{\mathrm{C}} \nu x\,(P \mid Q_i) \\
\nu x\,(P_1 \mid Q) &\longrightarrow_{\mathrm{C}} \nu x\,(P_2 \mid Q), \quad \text{if } P_1 \longrightarrow_{\mathrm{C}} P_2
\end{aligned}
$$

Commuting conversions

$$
\begin{aligned}
\nu z\,(x[y].(P \mid Q) \mid R) &\longrightarrow_{\mathrm{CC}} x[y].(\nu z\,(P \mid R) \mid Q), \qquad \text{if } z \notin fv(Q) \\
\nu z\,(x[y].(P \mid Q) \mid R) &\longrightarrow_{\mathrm{CC}} x[y].(P \mid \nu z\,(Q \mid R)), \qquad \text{if } z \notin fv(P) \\
\nu z\,(x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} x(y).\nu z\,(P \mid Q) \\
\nu z\,(x().P \mid Q) &\longrightarrow_{\mathrm{CC}} x().\nu z\,(P \mid Q) \\
\nu z\,(x[in_i].P \mid Q) &\longrightarrow_{\mathrm{CC}} x[in_i].\nu z\,(P \mid Q) \\
\nu z\,(\mathsf{case}\ x\ \{P; Q\} \mid R) &\longrightarrow_{\mathrm{CC}} \mathsf{case}\ x\ \{\nu z\,(P \mid R); \nu z\,(Q \mid R)\} \\
\nu z\,(\mathsf{case}\ x\ \{\} \mid Q) &\longrightarrow_{\mathrm{CC}} \mathsf{case}\ x\ \{\}
\end{aligned}
$$

Fig. 7: CP Congruences and Cut Reduction

3.2 Translation from CP to GV

In this section, we show that GV can simulate CP. Figure 8 gives the translation of CP into GV; typing environments are translated by the pointwise extension of the translation on types. We rely on our encoding of choice in GV (§2.1).

In translating CP terms to GV terms, the key observation is that CP terms contain their continuations; for example, the translation of input includes both a call to receive and the translation of the continuation. Additionally, the rebinding that is implicit in CP syntax is made explicit in GV. The translation $\mathcal{C}(\!|-|\!)$ translates top-level cuts to GV configurations; cuts that appear under prefixes are translated to applications of fork. As CP processes do not have return values, the translation of a CP process contains no main thread.

It is straightforward to see that the translation preserves typing; note that the channels in the CP typing environment become free variables in its translation.

Theorem 15. *If* $P \vdash \Delta$ *then* $(\!|\Delta|\!) \vdash^{\circ} \mathcal{C}(\!|P|\!)$.

Structural congruence in CP is a subset of the structural congruence relation for GV configurations; thus the translation trivially preserves congruence.

Theorem 16. *If* $P \equiv Q$, *then* $\mathcal{C}(\!|P|\!) \equiv \mathcal{C}(\!|Q|\!)$.

On types

$$(\!|A \otimes B|\!) = !\overline{(\!|A|\!)}.(\!|B|\!) \quad (\!|1|\!) = \mathsf{end}_! \quad (\!|A \oplus B|\!) = (\!|A|\!) \oplus (\!|B|\!) \quad (\!|0|\!) = \oplus\{\}$$
$$(\!|A \parr B|\!) = ?(\!|A|\!).(\!|B|\!) \quad (\!|\bot|\!) = \mathsf{end}_? \quad (\!|A \,\&\, B|\!) = (\!|A|\!) \,\&\, (\!|B|\!) \quad (\!|\top|\!) = \&\{\}$$

On terms

$$\begin{aligned}
(\!|\nu x\ (P \mid Q)|\!) &= \mathsf{let}\ x = \mathsf{fork}\ (\lambda x.(\!|P|\!))\ \mathsf{in}\ (\!|Q|\!) \\
(\!|x \leftrightarrow y|\!) &= \mathsf{link}\ (x, y) \\
(\!|x[y].(P \mid Q)|\!) &= \mathsf{let}\ x = \mathsf{send}\ (\mathsf{fork}\ (\lambda y.(\!|P|\!)), x)\ \mathsf{in}\ (\!|Q|\!) \\
(\!|x(y).P|\!) &= \mathsf{let}\ (y, x) = \mathsf{receive}\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|x[].0|\!) &= x \\
(\!|x().P|\!) &= \mathsf{let}\ () = \mathsf{wait}\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|x[in_1].P|\!) &= \mathsf{let}\ x = \mathsf{select}\ \mathsf{inl}\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|x[in_2].P|\!) &= \mathsf{let}\ x = \mathsf{select}\ \mathsf{inr}\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|\mathsf{case}\ x\ \{P; Q\}|\!) &= \mathsf{offer}\ x\ \{\mathsf{inl}\ x \mapsto (\!|P|\!); \mathsf{inr}\ x \mapsto (\!|Q|\!)\} \\
(\!|\mathsf{case}\ x\ \{\}|\!) &= \mathsf{let}\ (y, x) = \mathsf{receive}\ x\ \mathsf{in}\ \mathsf{absurd}\ y
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}(\!|\nu x\ (P \mid Q)|\!) &= (\nu x)(\mathcal{C}(\!|P|\!) \parallel \mathcal{C}(\!|Q|\!)) \\
\mathcal{C}(\!|P|\!) &= \circ\,(\!|P|\!), \quad P \text{ is not a cut}
\end{aligned}$$

Fig. 8: Translation of CP Terms into GV

Finally, observe that the translation of any prefixed CP term is a GV thread of either the form $F[K\ M]$ for $K \in \{\mathsf{send}, \mathsf{receive}, \mathsf{wait}\}$ or is $\circ x$ for some variable x. Thus, we can see that any cut reduction immediately possible for a process P is similarly possible for $(\!|P|\!)$. Following such a reduction, several additional GV reductions may be necessary to expose the next possible communication, such as substituting the received values into the continuation in the case of the translation of input, or spawning new threads in the translation of composition.

Theorem 17. *If $P \vdash \Delta$ and $P \longrightarrow_C Q$, then $\mathcal{C}(\!|P|\!) \longrightarrow^+ \mathcal{C}(\!|Q|\!)$.*

Proof. By induction on P; the cases are all straightforward.

The commuting conversions in CP do not expose additional reductions, but are only necessary to assure that the result of evaluation does not have a cut at the top level. Our characterisation of deadlock freedom in GV has no such requirement, so we have no need for corresponding steps in GV.

3.3 Translation from GV to CP

In this section, we show that CP can simulate GV. Figure 9 gives the translation on types and Figure 10 gives the translation on terms, substitutions, and configurations; we translate type environments pointwise on types.

The translation on session types is homomorphic except for output, where the output type is dualised. This accounts for the discrepancy between $\overline{!T.S} = ?T.S$ and $(A \otimes B)^\bot = A^\bot \parr B^\bot$. Following our previous work [19], the translation on functional types is factored through an auxiliary translation $[\![-]\!]$. The intuition

Session types

$$[\![!T.S]\!] = [\![T]\!]^{\perp} \otimes [\![S]\!] \qquad [\![?T.S]\!] = [\![T]\!] \parr [\![S]\!] \qquad [\![\mathsf{end}_!]\!] = 1 \qquad [\![\mathsf{end}_?]\!] = \bot$$

Functional types

$$[\![T]\!] = (\!|T|\!)^{\perp}, \quad \text{if } T \text{ is not a session type}$$

$$(\!|\mathbf{0}|\!) = 0 \qquad (\!|\mathbf{1}|\!) = 1$$
$$(\!|T + U|\!) = (\!|T|\!) \oplus (\!|U|\!) \qquad (\!|T \times U|\!) = (\!|T|\!) \otimes (\!|U|\!)$$
$$(\!|T \multimap U|\!) = (\!|T|\!)^{\perp} \parr (\!|U|\!)$$
$$(\!|S|\!) = [\![S]\!]$$

Fig. 9: Translation of GV Types into CP

is that the translation $[\![T]\!]$ of a functional type T is the type of its *interface*, whereas $(\!|T|\!)$ is the type of its *implementation*.

As CP processes do not have return values, the translation $[\![M]\!]z$ of a term M of type T includes the additional argument $z : [\![T]\!]^{\perp}$, which is a channel for simulating the return value. The translation on session terms is somewhat complicated by the need to include apparently trivial axiom cuts (highlighted in grey). These are needed to align with the translation of values, which permit further reduction inside the value constructors. The output in the translation of a fork arises from the need to apply the argument to a freshly generated channel (notice that application is simulated by an output). Linking is simulated by a link ($\leftrightarrow$) guarded by a nullary input which matches the nullary output of the output channel. Sending is simulated by output as one might expect. Receiving is simulated by input composed with sending the result to the return channel. Waiting is simulated by simply connecting the result to the return channel.

Variables are linked to the return channel. Closures are simulated by input, subject to an appropriate substitution, and application by output. Unit values are simulated by empty output to the return channel. Pairs are simulated by evaluating both components in parallel, transmitting the first along the return channel, and linking the second to the continuation of the return channel. Injections are simulated by injections. Each elimination form (other than application) guards the continuation with a suitable prefix, delaying reduction of the continuation until a value has been computed to pass to it. Substitutions are translated to right-nested sequences of cuts.

The translation of configurations is quite direct. We write $C \parallel_x C'$ to indicate that the variable x is shared by C and C'; in a well-typed GV configuration, there will always be exactly one such variable, so the translation is unambiguous.

Our translation differs from both Wadler's [26] and our previous one [19], neither of which simulate even plain β-reduction. This is because the obvious translation to CP cannot simulate substitution under a lambda abstraction, motivating our use of closures / weak explicit substitution. Indeed, others have taken advantage of full explicit substitutions in order simulate small-step semantics of λ-calculi in the full π-calculus [24].

Session terms

$$
\begin{aligned}
[\![\mathsf{fork}\ M]\!]z &= \nu w\,(w \leftrightarrow z \mid \nu x\,([\![M]\!]x \mid \nu y\,(x\langle w\rangle.x \leftrightarrow y \mid y[])))\\
[\![\mathsf{link}\ (M,N)]\!]z &= \nu v\,(v \leftrightarrow z \mid \nu w\,(v \leftrightarrow w \mid \nu x\,([\![M]\!]x \mid \nu y\,([\![N]\!]y \mid w().x \leftrightarrow y))))\\
[\![\mathsf{send}\ (M,N)]\!]z &= \nu x\,([\![N]\!]x \mid \nu y\,([\![M]\!]y \mid x\langle y\rangle.x \leftrightarrow z))\\
[\![\mathsf{receive}\ M]\!]z &= \nu y\,([\![M]\!]y \mid y(x).\nu w\,(w \leftrightarrow y \mid z\langle x\rangle.w \leftrightarrow z))\\
[\![\mathsf{wait}\ M]\!]z &= \nu y\,(y \leftrightarrow z \mid [\![M]\!]y)
\end{aligned}
$$

Functional terms

$$
\begin{aligned}
[\![x]\!]z &= x \leftrightarrow z\\
[\![\lambda^{\sigma} x.M]\!]z &= [\![\sigma]\!](z(x).[\![M]\!]z)\\
[\![L\,M]\!]z &= \nu x\,([\![M]\!]x \mid \nu y\,([\![L]\!]y \mid y\langle x\rangle.y \leftrightarrow z))\\
[\![()]\!]z &= z[]\\
[\![\mathsf{let}\ () = M\ \mathsf{in}\ N]\!]z &= \nu y\,([\![M]\!]y \mid y().[\![N]\!]z)\\
[\![(M,N)]\!]z &= \nu x\,([\![M]\!]x \mid \nu y\,([\![N]\!]y \mid z\langle x\rangle.y \leftrightarrow z))\\
[\![\mathsf{let}\ (x,y) = M\ \mathsf{in}\ N]\!]z &= \nu y\,([\![M]\!]y \mid y(x).[\![N]\!]z)\\
[\![\mathsf{inl}\ M]\!]z &= \nu x\,([\![M]\!]x \mid z[in_1].x \leftrightarrow z)\\
[\![\mathsf{inr}\ M]\!]z &= \nu x\,([\![M]\!]x \mid z[in_2].x \leftrightarrow z)\\
[\![\mathsf{case}\ L\ \{\mathsf{inl}\ x \mapsto M; \mathsf{inr}\ x \mapsto N\}]\!]z &= \nu x\,([\![L]\!]x \mid \mathsf{case}\ x\ \{[\![M]\!]z; [\![N]\!]z\})\\
[\![\mathsf{absurd}\ L]\!]z &= \nu x\,([\![L]\!]x \mid \mathsf{case}\ x\ \{\})
\end{aligned}
$$

Substitutions

$$[\![\{V_i/x_i\}]\!](P) = \hat{\nu}(x_i \mapsto [\![V_i]\!]x_i)_i[P]$$

$$\hat{\nu}(x_i \mapsto P_i)_i[P] \triangleq \nu x_1\,(P_1 \mid \ldots \nu x_n\,(P_n \mid P)\ldots)$$

Configurations

$$
\begin{aligned}
[\![\circ\, M]\!]z &= \nu y\,([\![M]\!]y \mid y[])\\
[\![\bullet\, M]\!]z &= [\![M]\!]z\\
[\![(\nu x)C]\!]z &= [\![C]\!]z\\
[\![C \parallel_x C']\!]z &= \nu x\,([\![C]\!]z \mid [\![C']\!]z)
\end{aligned}
$$

Fig. 10: Translation of GV Terms, Substitutions, and Configurations into CP

Another departure from the previous translations to CP is that, despite the call-by-value semantics of GV, our translation is more in the spirit of call-by-name. For instance, in the translation of an application $L\,M$, the evaluation of L and M can happen in parallel, and β-reduction can occur before M has reduced to a value. The previous translations hide the evaluation of M behind the prefix $y\langle x\rangle$, which means that reduction of M can get stuck in the case that L is a free variable. Short of performing a CPS transformation on the translation, our new approach seems necessary in order to ensure that $[\![-]\!]$ preserves reduction.

It is straightforward to show that the translation preserves typing.

Theorem 18.

1. *If* $\Gamma \vdash M : T$, *then* $[\![M]\!]z \vdash [\![\Gamma]\!], z : [\![T]\!]^{\perp}$.
2. *If* $\Gamma \vdash C$, *then* $\exists T.[\![C]\!]z \vdash [\![\Gamma]\!], z : [\![T]\!]^{\perp}$.

Proof. By induction on derivations. □

We now show that reduction in GV is preserved by reduction in CP. First, we observe that structural equivalence is preserved.

Theorem 19. *If* $\Gamma \vdash C$, $\Gamma \vdash D$, *and* $C \equiv D$, *then* $[\![C]\!]z \equiv [\![D]\!]z$.

Proof. By induction on the derivation of $\Gamma \vdash C$. □

As the translations on terms and configurations are compositional, we can mechanically lift them to translations on evaluation contexts and configuration contexts such that the following lemma holds by construction. Each translation of a context takes two arguments: a function that describes the CP term to plug into the hole, and an output channel.

Lemma 20. *For* $X \in \{E, F, G\}$, $[\![X[M]]\!]z = [\![X]\!][[\![M]\!]z$

We will make implicit use of Lemma 20 throughout our proofs. We write $x \mapsto P$ for a function that maps a name x to a process P that depends on x.

We now show that substitution commutes with $[\![-]\!]$.

Lemma 21. *If* $\Gamma \vdash M : T$, $\Gamma \vdash \sigma : \Delta$, *and* $z \notin dom(\sigma)$, *then* $[\![\sigma]\!]([\![M]\!]z) \Longrightarrow [\![M\sigma]\!]z$.

Proof. By induction on the structure of M. Here we show the cases for variables and closures.

- Case x. By linearity there exists V such that $\sigma = \{V/x\}$.

$$[\![\sigma]\!]([\![x]\!]z) = \nu x\,([\![V]\!]x \mid x \leftrightarrow z) \longrightarrow [\![V]\!]z = [\![x\sigma]\!]z$$

- Case $\lambda^{\sigma'}x.M$.

$$\begin{array}{ll} & [\![\sigma]\!]([\![\lambda^{\sigma'}x.M]\!]) \\ = & \quad (\sigma' = \{V_i/x_i\}_i) \\ & [\![\sigma]\!](\hat{\nu}(x_i \mapsto ([\![V_i]\!]x_i))_i[z(x).[\![M]\!]z]) \\ = & \quad (\sigma = \sigma_1 \uplus \cdots \uplus \sigma_n \text{ where } dom(\sigma_i) = fv(V_i)) \\ & [\![\sigma_1]\!](\ldots[\![\sigma_n]\!](\hat{\nu}(x_i \mapsto [\![V_i]\!]x_i)_i[z(x).[\![M]\!]z])) \\ = & \quad \text{(structural equivalence)} \\ & \hat{\nu}(x_i \mapsto [\![\sigma_i]\!]([\![V_i]\!]x_i))_i[z(x).[\![M]\!]z] \\ \Longrightarrow & \text{(IH)} \\ & \hat{\nu}(x_i \mapsto [\![V_i\sigma_i]\!]x_i)_i[z(x).[\![M]\!]z] \\ = & \quad (V_i\sigma_i = V\sigma) \\ & \hat{\nu}(x_i \mapsto [\![V_i\sigma]\!]x_i)_i[z(x).[\![M]\!]z] \\ = & \quad \text{(definition of } [\![-]\!]) \\ & [\![\lambda^{\sigma'\sigma}x.M]\!] \\ = & \quad \text{(definition of substitution)} \\ & [\![\lambda^{\sigma'}x.M\sigma]\!] \end{array}$$

Each of the remaining non-binding form cases follows straightforwardly using the induction hypothesis. Each of the remaining binding form cases requires a commuting conversion to push the appropriate substitution through a prefix. □

Using the substitution lemma, we prove that $[\![-]\!]$ preserves reduction on terms.

Theorem 22. *If $\Gamma \vdash M$, and $M \longrightarrow_V N$, then $[\![M]\!]z \Longrightarrow [\![N]\!]v$.*

Proof. By induction on the derivation of $M \longrightarrow_{\mathrm{V}} N$. Here we show the case of β-reduction.

– Case $(\lambda^{\sigma} x.M)\ V \longrightarrow_{\mathrm{V}} M(\{V/x\} \cup \sigma)$.

$$
\begin{array}{ll}
 & [\![(\lambda^{\sigma} x.M)\ V]\!]z \\
= & \quad \text{(definition of } [\![-]\!]) \\
 & \nu w\,([\![V]\!]w \mid \nu y\,([\![\sigma]\!](y(x).[\![M]\!]y) \mid y[x](w \leftrightarrow x \mid y \leftrightarrow z))) \\
\Longrightarrow_{\mathrm{C}} & \text{(cut send against receive)} \\
 & \nu w\,([\![V]\!]w \mid \nu y\,(y \leftrightarrow z \mid \nu x\,(w \leftrightarrow x \mid [\![\sigma]\!]([\![M]\!]y)))) \\
\Longrightarrow_{\mathrm{C}} & \text{(cut links and } \alpha \text{ rename)} \\
 & \nu x\,([\![V]\!]x \mid [\![\sigma]\!]([\![M]\!]z)) \\
\Longrightarrow & \quad \text{(by Lemma 21)} \\
 & [\![M(\{V/x\} \uplus \sigma)]\!]
\end{array}
$$

The remaining base cases are similarly direct. The inductive case for reduction inside an evaluation context follows straightforwardly by observing that the translation of an evaluation context never places its argument inside a prefix. □

Finally, we prove that $[\![-]\!]$ preserves reduction on configurations.

Theorem 23. *If $\Gamma \vdash C$, $\Gamma \vdash D$, and $C \longrightarrow D$, then $[\![C]\!]z \Longrightarrow [\![D]\!]z$.*

Proof. By induction on the derivation of $C \longrightarrow D$. The inductive cases follow straightforwardly from the compositionality of the definitions and Theorem 22. The details appear in Appendix A. □

4 Extending GV

In this section, we consider two variants of our core calculus: the first adopts a single self-dual type for closed channels; the second adds unlimited types.

4.1 Unifying $\mathsf{end}_!$ and $\mathsf{end}_?$

We begin by defining a language, based on GV, but combining the types $\mathsf{end}_!$ and $\mathsf{end}_?$ of closed channels. Figure 11 gives the alterations to the syntax and typing rules. The dual session types $\mathsf{end}_!$ and $\mathsf{end}_?$ are replaced by a single, self-dual type end; a new constant, close is provided to eliminate channels of type end. (In many existing systems, channels of type end are treated as unlimited, subject to weakening, rather than requiring an explicit close. We have left close explicit to simplify the presentation.) The type schemas for fork and link have been simplified, as we no longer need to build elimination of closed channels into fork. Figure 12 gives the updated evaluation rules for the extended language. In

Syntax

$$\begin{array}{lll} & \text{Session types} & S ::= !T.S \mid ?T.S \mid \mathsf{end} \mid S^\sharp \\ & \text{Constants} & K ::= \mathsf{send} \mid \mathsf{receive} \mid \mathsf{fork} \mid \mathsf{close} \mid \mathsf{link} \end{array}$$

Changes to duality

$$\overline{\mathsf{end}} = \mathsf{end}$$

Changes to type schemas for constants

$$\mathsf{fork} : (S \multimap \mathbf{1}) \multimap \overline{S} \qquad \mathsf{close} : \mathsf{end} \multimap \mathbf{1} \qquad \mathsf{link} : S \times \overline{S} \multimap \mathbf{1}$$

Fig. 11: Syntax and Typing Rules for Combined Closed Channels

Extended configuration equivalence

$$C \parallel \circ () \equiv C$$

Extended reduction rules (all other reduction rules apply as in GV)

CLOSE

$$\overline{(\nu x)(F[\mathsf{close}\, x] \parallel F'[\mathsf{close}\, x]) \longrightarrow F[()] \parallel F'[()]}$$

LINK

$$\overline{(\nu x)(F[\mathsf{link}\, (x, y)] \parallel C) \longrightarrow F[()] \parallel C\{y/x\}}$$

Fig. 12: Updated Configuration Evaluation Rules

addition to a new rule for close (replacing the one for wait), the rule for link can be simplified (as it can now return a unit value instead of a closed channel).

Our modified language is, perhaps surprisingly, strictly more expressive than GV. Consider the following term:

$$\mathsf{let}\ w = \mathsf{fork}\, (\lambda w.\mathsf{close}\, w; M)\ \mathsf{in}\ \mathsf{close}\, w; N$$

Initially, the forked thread and its parent share channel w. After both threads close w, there can be no further communication between the threads; in contrast, in core GV, there must always be a final synchronisation with wait. To account for the increase in expressivity, we must extend the existing configuration typing rules (Figure 5) with a rule for composition in which no channels are shared:

$$\frac{\Gamma \vdash^{\phi} C \quad \Gamma' \vdash^{\phi'} C'}{\Gamma, \Gamma' \vdash^{\phi+\phi'} C \parallel C'}$$

Despite the additional expressivity of the modified calculus, we might hope that our results on deadlock freedom and progress (Theorems 7 and 10) would apply to this calculus as well. For the modified calculus, we must adapt Lemma 5:

Lemma 5A. *If $\Gamma \vdash C$ and $C = G[D \parallel D']$, then $fv(D) \cap fv(D')$ is either empty or the singleton set $\{x\}$ for some variable x.*

Syntactic extensions

Types	$T ::= \Box T \mid \ldots$
Terms	$M, N ::= \mathsf{let}\ !x = M\ \mathsf{in}\ N \mid\ !M \mid \ldots$
Values	$V ::= !^{\sigma} E \mid \ldots$
Evaluation contexts	$E ::= \mathsf{let}\ !x = E\ \mathsf{in}\ M \mid \ldots$

Typing rules

$$\frac{\Gamma \vdash M : T \quad \Box\Gamma}{\Gamma \vdash\ !M : \Box T} \qquad \frac{\Gamma \vdash M : \Box T \quad \Gamma', x : T \vdash N : U}{\Gamma \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N : U}$$

$$\frac{\Gamma \vdash M : T}{\Gamma, x : \Box U \vdash M : T} \qquad \frac{\Gamma, x : \Box T, x' : \Box T \vdash M : U}{\Gamma, x : \Box T \vdash M\{x/x'\} : T}$$

Reduction

$$\mathsf{let}\ !x = !^{\sigma} M\ \mathsf{in}\ N \longrightarrow_{\mathrm{V}} N\{(M\sigma)/x\}$$

Fig. 13: GV Extensions for Unlimited Types

Clearly, this change does not allow the introduction of cyclic dependencies. Thus, the adaptation of the deadlock freedom and progress results to the modified calculus is entirely mechanical. It is straightforward to show that the other theorems of (§2.2) still hold in the presence of a single self-dual type for closed channels.

The additional expressivity does mean that we cannot define a translation from the modified calculus to CP. We believe that we could do so were CP extended with terms corresponding to the mix rules:

$$\frac{}{0 \vdash} \qquad \frac{P \vdash \Delta \quad Q \vdash \Delta'}{P \mid Q \vdash \Delta, \Delta'}$$

4.2 Unlimited Types

So far, we have treated only linear types. In this section, we consider one standard approach to extending the term language to include unlimited types.

Figure 13 gives the extension of GV. We begin by adding a new class of types, $\Box T$, representing unlimited types. (The typical notation for such types in linear logic, $!T$, clashes with the notation for output in session types.) We add terms to construct and deconstruct values of type $\Box T$; $\Box\Gamma$ denotes that every type in Γ must be of the form $\Box U$ for some type U. Values of type $\Box T$ can be weakened (discarded) and contracted (duplicated). We extend the language of values with unlimited values $!^{\sigma} E$; note that, as an unlimited value behaves similarly to a closure, we must introduce an explicit substitution. As in the treatment of λ-terms, we extend the typing relation to take account of the substitution

$$\frac{\Gamma \vdash M\sigma : T \quad dom(\sigma) = fv(M) \quad \Box\Gamma}{\Gamma \vdash\ !^{\sigma} M : \Box T}$$

and treat a term $!M$ as an abbreviation as follows:

$$!M \triangleq !^{\sigma}(M\sigma') \quad \text{where } fv(M) = \{x_1, \ldots, x_n\} \quad y_1, \ldots, y_n \text{ are fresh variables}$$
$$\sigma = \{x_1/y_1, \ldots, x_n/y_n\} \qquad \sigma' = \{y_1/x_1, \ldots, y_n/x_n\}$$

The reduction rule for $\Box T$ values is unsurprising—however, unlike in the other reductions, x may be used non-linearly in M. As the concurrent semantics is unchanged from the base calculus, the extension of deadlock freedom and progress to this calculus is mechanical. Similarly, it is not difficult to show that the other theorems of (§2.2) still hold in the presence of either or both extensions. The only non-trivial feature is the need for a logical relations argument in order to prove strong normalisation in the presence of unlimited types.

Appendix B extends CP with replication (following Wadler [26]) and correspondingly adapts the translations between CP and GV.

5 Related Work

Session Types and Functional Languages. Session types were originally proposed by Honda [13], and later extended by Takeuchi et al. [22] and by Honda et al. [14]. Honda's system relies on a substructural type system (in which channels cannot be duplicated or discarded) and adopts the syntax $\&$ and $\oplus$ for choice; however, he does not draw a connection between his type system and the connectives of linear logic, and his system includes a single, self-dual closed channel. Vasconcelos et al. [25] develop a language that integrates session-typed communication primitives and a functional language. Gay and Vasconcelos [10] extend the approach to describe asynchronous communication with statically-bounded buffers. Their approach provides a more flexible mechanism of session initiation, distinct from their construct for thread creation, and they do not consider deadlock. Kobayashi [15] describes an embedding of session-typed π-calculus in polyadic linear π-calculus, relying on multi-argument send and receive to capture the state of a communication and variant types to capture choice; Dardha et al. [9] extend his approach to subtyping and polymorphism.

Linear Logic and Session Types. When he originally described linear logic, Girard [12] suggested that it would be suited to reasoning about concurrency. Abramsky [1] and Bellin and Scott [3] give embeddings of linear logic proofs in π-calculus, and show that cut reduction is simulated by π-calculus reduction. Their work is not intended to provide a type system for π-calculus: there are many processes which are not the image of some proof.

Caires and Pfenning [5] present a session type system for π-calculus that exactly corresponds to the proof system for the dual intuitionistic linear logic, and show that (up to congruence) cut reductions corresponds to process reductions or process equivalences. Toninho et al. [23] consider embeddings of the λ-calculus into session-typed π-calculus; their focus is on expressing the concurrency inherent in λ-calculus terms, rather than simulating standard reduction. Wadler [26]

adapts the approach of Caires and Pfenning to classical (rather than intuitionistic) linear logic, and gives a translation from GV (his functional language) to CP (his process calculus). He does not give a direct semantics for GV. In previous work [19], we give a type-preserving translation from CP to GV.

Deadlock Freedom and Progress. There have been several approaches to guarantee deadlock freedom in π-calculus. Kobayashi [16] and Padovani [20] extend type systems for linear π-calculus with priority information, capturing the order in which channels are used. Giachino et al. [11] give a type system that expresses dependencies directly in the types of CCS terms. These systems permit more programs than ours, at the cost of significantly more complex type systems; they also do not enjoy the close correspondence with linear logic (or other well-known logical systems).

Carbone and Debois [7] give a graphical characterisation of session-typed processes; this allows them to directly identify cycles in channel dependencies. They show that all possible interactions eventually take place in cycle-free processes. Carbone et al. [6] show similar results for well-typed processes under Kobayashi's type system for deadlock freedom; their approach accommodates processes with open channels by defining a type-directed closure of a process, and showing that open processes progress only if their typed closures progress.

6 Conclusion and Future Work

We have presented a small-step operational semantics for GV, a session-typed functional core language. We have proved that it is deadlock-free, deterministic, and terminating, and have established simulations both ways between our semantics for GV and cut-reduction in a process calculus based on linear logic. Furthermore, we have shown that GV provides a promising basis for future modular language development by illustrating two extensions to GV, both of which preserve deadlock-freedom, determinism, and termination.

We identify two important directions for future work: recursion and asynchronous communication. Recursion is essential both for channels (to capture repeating behaviour, such as adding recipients to a mail message) and for functional programming. Adding unchecked recursion to GV would clearly compromise termination and introduce the possibility of livelock; we hope that adapting approaches used for fixed points in linear logic might mitigate this issue. Asynchronous communication naturally lends itself to practical implementation. We hope to develop the approach of Gay and Vasconcelos [10] and show a correspondence between synchronous and asynchronous semantics for GV.

Acknowledgements. Thanks to Philip Wadler and the anonymous reviewers. This work was funded by EPSRC grant number EP/K034413/1.

References

[1] S. Abramsky. Proofs as processes. *Theor. Comput. Sci.*, 135(1):5–9, Apr. 1992.

[2] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.

[3] G. Bellin and P. J. Scott. On the π-Calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

[4] M. Boreale. On the expressiveness of internal mobility in name-passing calculi. In *CONCUR*, pages 163–178. Springer, 1996.

[5] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236. Springer, 2010.

[6] M. Carbone, O. Dardha, and F. Montesi. Progress as compositional lock-freedom. In *COORDINATION 2014*, pages 49–64. Springer, 2014.

[7] M. Carbone and S. Debois. A graphical approach to progress for structured communication in web services. In *ICE*, pages 13–27, 2010.

[8] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *LNCS*. Springer, 2006.

[9] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP*, pages 139–150. ACM, 2012.

[10] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50, 2010.

[11] E. Giachino, N. Kobayashi, and C. Laneve. Deadlock analysis of unbounded process networks. In *CONCUR*, pages 63–77. Springer, 2014.

[12] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, Jan. 1987.

[13] K. Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523. Springer, 1993.

[14] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138. Springer, 1998.

[15] N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, pages 439–453. Springer, 2002.

[16] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, pages 233–247. Springer, 2006.

[17] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371. ACM, 1996.

[18] J. Lévy and L. Maranget. Explicit substitutions and programming languages. In *Foundations of Software Technology and Theoretical Computer Science, 1999*, volume 1738 of *LNCS*, pages 181–200. Springer, 1999.

[19] S. Lindley and J. G. Morris. Sessions as propositions. In *PLACES*, 2014.

[20] L. Padovani. Deadlock and lock freedom in the linear π-calculus. In *LICS*, page 72. ACM, 2014.

[21] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Inf. Comput.*, 239:254–302, 2014.

[22] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413. Springer, 1994.

[23] B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. In *FOSSACS*, pages 346–360. Springer, 2012.

[24] S. van Bakel and M. G. Vigliotti. A logical interpretation of the λ-calculus into the π-calculus, preserving spine reduction and types. In *CONCUR*, pages 84–98. Springer, 2009.

[25] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[26] P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

A Selected Proofs

Theorem 2. *If $\Gamma \vdash C$ and $C \longrightarrow C'$ then $\Gamma \vdash C'$.*

Proof. By induction on the derivation of $C \longrightarrow C'$. We include several representative cases.

- Case Lift is immediate by the induction hypothesis.
- Case LiftV follows from Lemma 1.
- In case Send, from the assumption $\Gamma \vdash (\nu x)(F[\mathsf{send}\,(V, x)] \parallel F'[\mathsf{receive}\,x])$, we have $\Gamma, x : S^\sharp \vdash F[\mathsf{send}\,(V, x)] \parallel F'[\mathsf{receive}\,x]$, from which we can assume that Γ partitions as Γ_1, Γ_2 such that $S = !T.S'$, V has type T, and $\Gamma_1, x : !T.S' \vdash F[\mathsf{send}\,(V, x)]$, $\Gamma_2, x : ?T.\overline{S'} \vdash F'[\mathsf{receive}\,x]$. As $\mathsf{send} : T \times !T.S' \multimap S'$ and $\Gamma_1, x : !T.S' \vdash F[\mathsf{send}\,(V, x)]$, we can conclude that $\Gamma_1, x : S' \vdash F[x]$. By a similar argument, we conclude that $\Gamma_2, x : \overline{S'} \vdash F'[(V, x)]$. Finally, we can recompose the resulting processes concluding that $\Gamma_1, \Gamma_2, x : S^\sharp \vdash F[x] \parallel F'[(V, x)]$ and hence $\Gamma \vdash (\nu x)(F[x] \parallel F'[(V, x)])$.
- In case Fork, from the assumption $\Gamma \vdash F[\mathsf{fork}\,V]$, we can conclude that Γ splits as Γ_1, Γ_2 and there is some S such that $\Gamma_1, x : S \vdash F[x]$ and $\Gamma_2, x : \overline{S} \vdash V\,x$. Thus we have that $\Gamma, x : S \vdash F[x] \parallel V\,x$ and $\Gamma \vdash (\nu x)(F[x] \parallel V\,x)$.
- In case Wait, from the assumption $\Gamma \vdash (\nu x)(F[\mathsf{wait}\,x] \parallel x)$, we can conclude that $\Gamma, x : \mathsf{end}_? \vdash F[\mathsf{wait}\,x]$ and thus, from the typing of wait, that $\Gamma \vdash F[()]$.
- In case Link, from the assumptions $\Gamma \vdash (\nu x)(F[\mathsf{link}\,(x, y)] \parallel F'[M])$, $x \in fv(M)$, we can conclude that Γ partitions as $\Gamma_1, \Gamma_2, y : S$ such that $\Gamma_1, y : S, x : \overline{S} \vdash F[\mathsf{link}\,(x, y)]$ and $\Gamma_2, x : S \vdash F'[M]$. (Note that the free variable assumption on the reduction rule for link allows us to assume that neither F nor F' binds x or y.) From the type of link, we have that $\Gamma, x : \mathsf{end}_! \vdash F[x]$; similarly, from $x \in fv(M)$, we can conclude that $\Gamma_2, y : S \vdash F'[M\{y/x\}]$. Finally, from the typing rule for wait, we have that $\Gamma_2, x : \mathsf{end}_?, y : S \vdash F'[\mathsf{wait}\,x; M\{y/x\}]$, and that $\Gamma \vdash (\nu x)(F[x] \parallel F'[\mathsf{wait}\,x; M\{y/x\}])$ □

Lemma 9. *If $\Gamma \vdash C$, then there is some $C' \equiv C$ such that $\Gamma \vdash C'$ and C' is in canonical form.*

Proof. Let $x_1, \ldots, x_{n-1}$ be the ν-bound variables in C and $M_1, \ldots, M_n$ be the terms in P; the proof is by induction on n. If $n > 1$, then pick some M_i such that there is exactly one ν-bound variable x_j where $x_j \in fv(M_i)$. (That there must be such an M_i and x_j can be established by a standard counting argument, together with Lemma 5.) Now, construct D from C by the homomorphic extension of the mapping $(\nu x_j)E \mapsto E; E \parallel \phi M_i \mapsto E$. From the assumption that $\Gamma \vdash C$, we can conclude that there is some $\Gamma' \subseteq \Gamma$ and type S such that $\Gamma', x_j : S \vdash D$. By the induction hypothesis, there is some $D' \equiv D$ in canonical form. Finally, let $C' = (\nu x_j)(\phi M_i \parallel D')$; we can see that straightforwardly that C' is in canonical form; that $C \equiv C'$; and, that $\Gamma \vdash C'$. □

Theorem 23. *If $\Gamma \vdash C$ and $C \longrightarrow D$, then $[\![C]\!]z \Longrightarrow [\![D]\!]z$.*

Proof. By induction on the derivation of $C \longrightarrow D$. The inductive cases follow straightforwardly from the compositionality of the definitions and Theorem 22.

– Case $(\nu x)(F[\mathsf{send}\,(V, x)] \parallel F'[\mathsf{receive}\,x]) \longrightarrow (\nu x)(F[x] \parallel F'[(V, x)])$.

$$
\begin{array}{ll}
& [\![(\nu x)(F[\mathsf{send}\,(V, x)] \parallel F'[\mathsf{receive}\,x])]\!]z \\
= & \\
& \nu x\,([\![F]\!][[\![\mathsf{send}\,(V, x)]\!]y]z \mid [\![F']\!][[\![\mathsf{receive}\,x]\!]]z) \\
= & \\
& \nu x([\![F]\!][z \mapsto \nu v\,([\![V]\!]v \mid \nu w\,(x \leftrightarrow w \mid w\langle v\rangle.w \leftrightarrow z))]z \\
& \quad \mid [\![F']\!][z \mapsto \nu y\,(x \leftrightarrow y \mid y(v).\nu w\,(y \leftrightarrow w \mid z\langle v\rangle.w \leftrightarrow z))]z \\
\Longrightarrow_{\mathrm{C}} & \text{(cut links)} \\
& \nu x([\![F]\!][z \mapsto \nu v\,([\![V]\!]v \mid x\langle v\rangle.x \leftrightarrow z)]z \\
& \quad \mid [\![F']\!][z \mapsto x(v).\nu w\,(x \leftrightarrow w \mid z\langle v\rangle.w \leftrightarrow z)]z \\
\Longrightarrow_{\mathrm{C}} & \text{(cut send against receive)} \\
& \nu x\,([\![F]\!][z \mapsto \nu v\,([\![V]\!]v \mid x \leftrightarrow z)] \mid [\![F']\!][z \mapsto \nu w\,(x \leftrightarrow w \mid z\langle v\rangle.w \leftrightarrow z)]z) \\
\equiv & \\
& \nu x\,([\![F]\!][z \mapsto x \leftrightarrow z]z \mid [\![F']\!][z \mapsto \nu v\,([\![V]\!]v \mid \nu w\,(x \leftrightarrow w \mid z\langle v\rangle.w \leftrightarrow z))]z) \\
= & \\
& [\![(\nu x)(F[x] \parallel F'[(V, x)])]\!]
\end{array}
$$

– Case $(\nu x)(F[\mathsf{wait}\,x] \parallel \circ x) \longrightarrow F[()]$.

$$
\begin{array}{ll}
& [\![(\nu x)(F[\mathsf{wait}\,x] \parallel \circ x)]\!]z \\
= & \\
& \nu x\,([\![F]\!][y \mapsto \nu w\,(w \leftrightarrow x \mid w \leftrightarrow y)]z \mid \nu y\,(x \leftrightarrow y \mid y[])) \\
\Longrightarrow_{\mathrm{C}} & \text{(cut links)} \\
& \nu x\,([\![F]\!][y \mapsto x \leftrightarrow y]z \mid x[]) \\
\Longrightarrow_{\mathrm{C}} & \text{(cut link)} \\
& [\![F]\!][y \mapsto y[]]z \\
= & \\
& [\![F[()]]\!]z
\end{array}
$$

– Case $F[\mathsf{fork}\,(\lambda^{\sigma}x.M)] \longrightarrow (\nu x)(F[x] \parallel M\sigma)$.

$$
\begin{array}{ll}
& [\![F[\mathsf{fork}\,(\lambda^{\sigma}x.M)]]\!]z \\
= & \\
& [\![F]\!][[\![\mathsf{fork}\,(\lambda^{\sigma}x.M)]\!]]z \\
= & \\
& [\![F]\!][z \mapsto \nu x\,([\![x]\!]z \mid \nu y\,([\![\sigma]\!](y(x).[\![M]\!]y) \mid \nu w\,(y\langle x\rangle.y \leftrightarrow w \mid w[])))]z \\
\Longrightarrow_{\mathrm{C}} & \text{(cut send against receive)} \\
& [\![F]\!][z \mapsto \nu x\,([\![x]\!]z \mid \nu y\,([\![\sigma]\!]([\![M]\!]y) \mid y[]))]z \\
\equiv & \\
& \nu x\,([\![F]\!][[\![x]\!]]z \mid [\![\sigma]\!]([\![\nu y\,([\![M]\!]y \mid y[])]\!])) \\
= & \\
& [\![(\nu x)(F[x] \parallel M\sigma)]\!]z
\end{array}
$$

– Case $(\nu x)(F[\mathsf{link}\,(x,y)] \parallel F'[M]) \longrightarrow (\nu x)(F[x] \parallel F'[\mathsf{wait}\,x; M\{y/x\}])$.

$$
\begin{aligned}
&[\![(\nu x)(F[\mathsf{link}\,(x,y)] \parallel F'[M])]\!]z \\
=\;& \\
&\nu x\,([\![F]\!][[\![\mathsf{link}\,(x,y)]\!]]z \mid [\![F']\!][[\![M]\!]]z) \\
=\;& \\
&\nu x([\![F]\!][z \mapsto \nu v(v \leftrightarrow z \mid \nu w(v \leftrightarrow w \mid \\
&\quad \nu x'\,(x \leftrightarrow x' \mid \nu y'\,(y \leftrightarrow y' \mid w().x' \leftrightarrow y'))))]z \mid [\![F']\!][[\![M]\!]]z) \\
\Longrightarrow_{\mathrm{C}}\;& (\text{cut links}) \\
&\nu x\,([\![F]\!][z \mapsto \nu v\,(v \leftrightarrow z \mid \nu w\,(v \leftrightarrow w \mid w().x \leftrightarrow y))]z \mid [\![F']\!][[\![M]\!]]z) \\
\equiv\;& \\
&\nu v\,([\![F]\!][z \mapsto v \leftrightarrow z]z \mid \nu x\,(\nu w\,(v \leftrightarrow w \mid w().x \leftrightarrow y) \mid [\![F']\!][[\![M]\!]]z)) \\
\equiv\;& \\
&\nu v\,([\![F]\!][z \mapsto v \leftrightarrow z]z \mid [\![F']\!][z \mapsto \nu w\,(v \leftrightarrow w \mid w().\nu x\,(x \leftrightarrow y \mid [\![M]\!]z))]z) \\
=\;& \\
&[\![(\nu x)(F[x] \parallel F'[\mathsf{wait}\,x; M\{y/x\}])]\!]z
\end{aligned}
$$

B Replication

Wadler's CP calculus provides replicated channels, used to obtain arbitrarily many copies of some concurrent behaviour, corresponding to the exponentials in linear logic. Figure 14 gives the typing and reduction rules for replicated channels; $?\Delta$ denotes a context in which all types are of the form $?A$ for some type A. Note that duplication and discarding of replicated processes happens in the cut reduction rules for weakening and contraction, not as part of the rule for dereliction. The rules for exponentials closely parallel the rules for unlimited GV values; thus, the extension of our translation from GV to CP to include unlimited values is straightforward.

$$
\begin{aligned}
[\![\Box T]\!] &\triangleq\; ![\![T]\!] \\
[\![!M]\!]z &\triangleq\; !z(y).[\![M]\!]y \\
[\![\mathsf{let}\;!x = V\;\mathsf{in}\;M]\!]z &\triangleq\; \nu y\,([\![V]\!]y \mid ?y[x].[\![M]\!]z)
\end{aligned}
$$

Theorem 24. *If $\Gamma \vdash C$, $\Gamma \vdash D$, and $C \longrightarrow D$, then $[\![C]\!]z \Longrightarrow [\![D]\!]z$.*

The translation in the other direction is not quite as simple: we must provide replicated channels, not just replicated values. However, following a similar pattern to our encoding of session-level choice using value-level sums, we can encode such channels using value-level unlimited values. First, we introduce new, dual session type constructors $\mathit{Service}(S)$ and $\mathit{Server}(S)$, defined by

$$
\mathit{Server}(S) \triangleq\; !(\Box\overline{S}).\mathsf{end}_! \qquad\qquad \mathit{Service}(S) \triangleq\; ?(\Box S).\mathsf{end}_?
$$

Note that $\overline{\mathit{Server}(S)} = \mathit{Service}(\overline{S})$. We then introduce new constants $\mathsf{replicate}$ and $\mathsf{request}$, with type signatures

$$
\mathsf{replicate} : (\mathit{Server}(S), \Box(S \multimap \mathsf{end}_!)) \multimap \mathsf{end}_! \qquad\qquad \mathsf{request} : \mathit{Service}(S) \multimap S
$$

Syntax

$$\text{Types } A, B ::= !A \mid ?A \mid \ldots$$
$$\text{Terms } P, Q ::= x(y).P \mid x[y].P \mid \ldots$$

Duality

$$(!A)^{\perp} = ?(A^{\perp}) \qquad (!A)^{\perp} = !(A^{\perp})$$

Typing rules

$$\frac{P \vdash ?\Delta, x : A}{x(y).P \vdash ?\Delta, x : !A} \qquad \frac{P \vdash \Delta, x : A}{x[y].P \vdash \Delta, x : ?A} \qquad \frac{P \vdash \Delta}{P \vdash \Delta, x : ?A}$$

$$\frac{P \vdash \Delta, x : ?A, x' : ?A}{P\{x/x'\} \vdash \Delta, x : ?A}$$

Primary cut reduction rules

$$\begin{aligned} \nu x\,(x(y).P \mid x[y].Q) &\longrightarrow_{\mathrm{C}} \nu y\,(P \mid Q) \\ \nu x\,(x(y).P \mid Q) &\longrightarrow_{\mathrm{C}} Q \qquad x \notin fv(Q) \\ \nu x\,(x(y).P \mid Q\{x/x'\}) &\longrightarrow_{\mathrm{C}} \nu x\,(x(y).P \mid \nu x'\,(x'(y).P \mid Q)) \end{aligned}$$

Commuting conversions

$$\begin{aligned} \nu z\,(x[y].P \mid Q) &\longrightarrow_{\mathrm{CC}} x[y].\nu z\,(P \mid Q) \\ \nu z\,(x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} x(y).\nu z\,(P \mid Q) \end{aligned}$$

Fig. 14: Replicated channels in CP

defined as follows:

$$\begin{aligned} \mathsf{replicate}(x, f) &\triangleq \mathsf{send}\,(!(\mathsf{let}\ !g = f\ \mathsf{in}\ \mathsf{fork}\ g), x) \\ \mathsf{request}\ s &\triangleq \mathsf{let}\ (w, s) = \mathsf{receive}\ s\ \mathsf{in}\ \mathsf{wait}\ s; \mathsf{let}\ !y = w\ \mathsf{in}\ y \end{aligned}$$

These can be used to define the translation from CP to GV:

$$\begin{aligned} (\!|!A|\!) &\triangleq \mathit{Server}((\!|A|\!)) \qquad (\!|?A|\!) \triangleq \mathit{Service}((\!|A|\!)) \\ (\!|!x(y).P|\!) &\triangleq \mathsf{replicate}(x, !(\lambda y.(\!|P|\!))) \\ (\!|?x[y].P|\!) &\triangleq \mathsf{let}\ y = \mathsf{request}\ x\ \mathsf{in}\ (\!|P|\!) \\ \left(\!\!\left| \frac{P \vdash \Delta}{P \vdash \Delta, x : ?A} \right|\!\!\right) &\triangleq \begin{array}{l} \mathsf{let}\ (w, x) = \mathsf{receive}\ x\ \mathsf{in} \\ \mathsf{wait}\ x; (\!|P|\!) \end{array} \\ \left(\!\!\left| \frac{P \vdash \Delta, x : ?A, y : ?A}{P\{x/z, y/z\} \vdash \Delta, z : ?A} \right|\!\!\right) &\triangleq \begin{array}{l} \mathsf{let}\ (w, z) = \mathsf{receive}\ z\ \mathsf{in}\ \mathsf{wait}\ z; \\ \mathsf{let}\ x = \mathsf{fork}\,(\lambda x.\mathsf{send}\,(w, x))\ \mathsf{in} \\ \mathsf{let}\ y = \mathsf{fork}\,(\lambda y.\mathsf{send}\,(w, y))\ \mathsf{in}\ (\!|P|\!) \end{array} \end{aligned}$$

The translation emphasises that, while weakening and contraction are implicit in CP, they play a central role in the CP semantics, and thus have non-trivial translations to GV. Theorem 17 directly extends to CP with replicated channels.

Theorem 25. *If $P \vdash \Delta$ and $P \longrightarrow_C Q$, then $\mathcal{C}(\!|P|\!) \longrightarrow^{+} \mathcal{C}(\!|Q|\!)$.*